Universal Content Launcher

A series of tutorials detailing the creation of a content launcher in Microsoft Visual C#

Tutorial Three

Launching the Content

Contents

Introduction 3
Tutorial Requirements 3
Part One: Launching the content 4
Part Two: Redefining the UCLContent class 12
Part Three: Debugging your code 20
Part Four: Ready to Launch 23

Introduction

In the tutorial two, we learned how to create a new class to hold the data we need to launch content off our hard drive, how to create an XML file to store the data, and how to read the XML file and use it to populate our main treeview display.

We also learned how to create a method, how to trap errors with `try` and `catch`, how to assign new events to a form and how to create a custom Type called an `enum`. We covered quite a lot, but still have little to show for it, so let's crack on with this tutorial and get some content launching done!

Tutorial Requirements

These tutorials are sequential, so if you haven't already been through the previous tutorials, then please download, and work through them before continuing with this one.

Tutorial one also outlines the software you will need for creating UCL and where you can download it for free.

Part One: Launching the content

We have come a long way since starting this project, right to the point where we can now see content in our content list in UCL! But how can we actually launch it?

Well, as it's in a treeview control, then it's safe to assume that most people will be expecting to double click on the selected content to launch it. There are also probably a few people that would like to right click on it and launch it from a popup menu, often referred to as a context menu.

Let's have a look at the double click method, as this is the most basic way we can get our content to launch. We're going to need to add a new method to our treeview control to handle whenever we double click on it, so double click on the treeview control to create a new method, like we have done in the past for menu items, and we will get this:

```
        private void treeView1_AfterSelect(object sender, TreeViewEventArgs
e)
        {

        }
```

Well, I'm not being funny or anything, but we wanted to do something when we double click on the treeview control, not after we have just clicked once to select the content, which is what this method is for. If we used this, then our content would launch as soon as someone clicked on it!

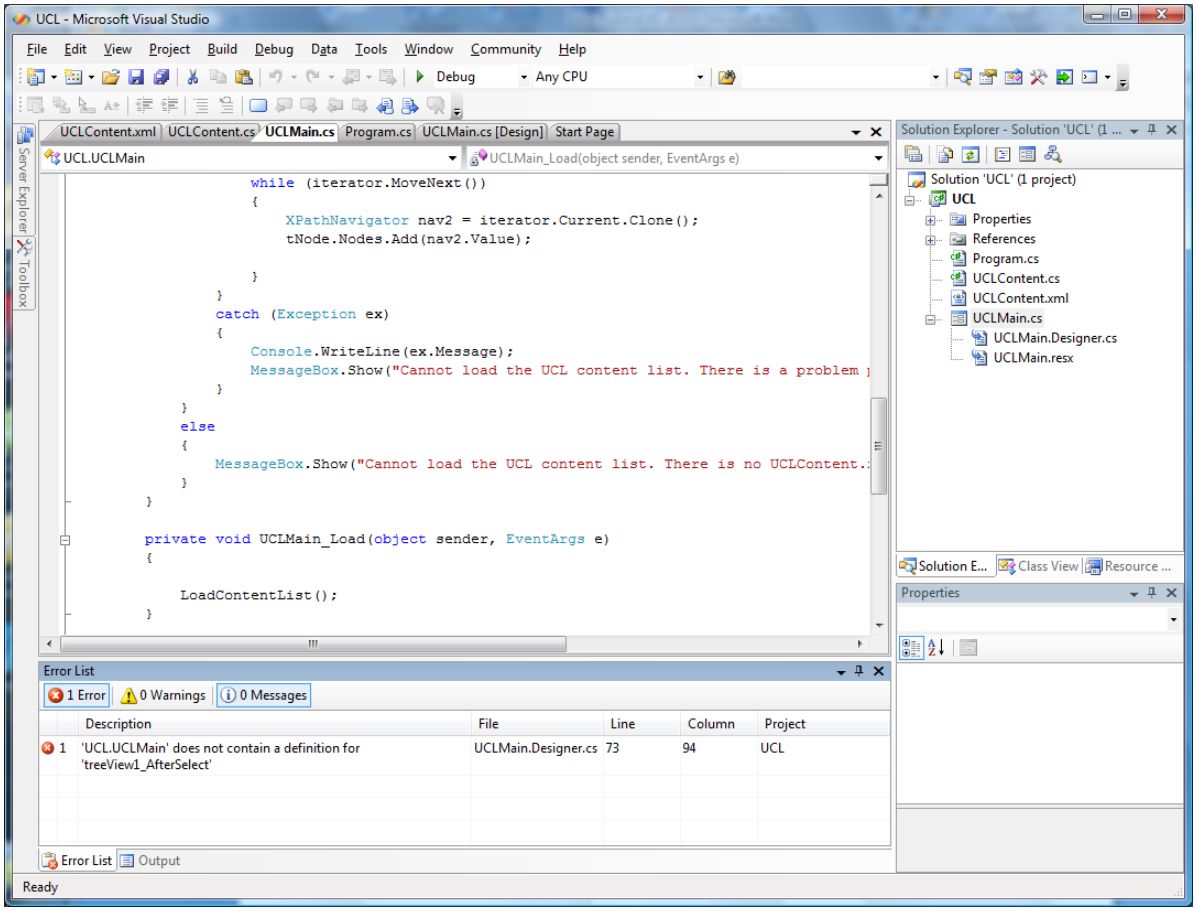
So, here's our first example of a (usually) helpful IDE getting it wrong! Of course the IDE can't read our mind, so it's only programmed to create a default method for the most common methods of any given control, anything else and we need to do a bit more work than just double clicking on the control in the design form and then going to the code that is generated for us.

Since we don't want this code, then delete it from the code window, and now I want you to try and run UCL:

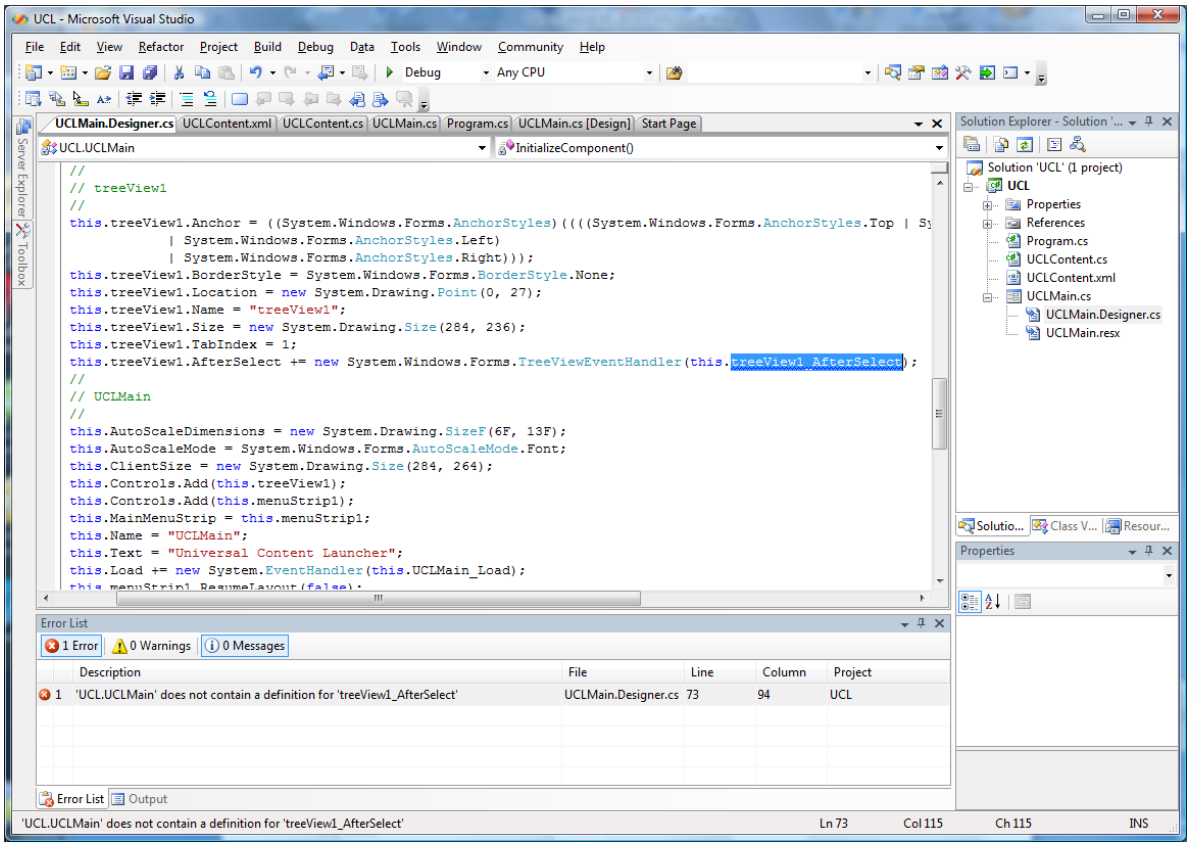
Oh dear – we broke our code and we're not even on page two! If you don't see the error box above, then you must have ticked that "Do not show this dialogue again" box at some point, and should be at the next screenshot on the next page, but please do read on!

Now, remember how I told you way back in tutorial one that our IDE creates code for us that can also be hidden away in other code modules as references? This is exactly what has happened here, and as we manually removed the code that was sat in front of us, we broke the link to some other code that the IDE has created somewhere. But where is it?

Click on No in the error dialogue and you should see the following:



Now then, remember when we got our first compiler error? We saw the error in the output window at the bottom, and also saw the code that was at fault in the main window. This time, the bad code is nowhere to be seen, but fear not, just double click on the error description in the output window to take you to the code.



Ah, what is all that? Well, once again, in tutorial one we noticed two files that are in our solution explorer under UCLMain.cs that I told you to ignore, and here's one of them come back to haunt us! If I explain what this file actually is, then it becomes apparent why we got this error.

For every form you design there will be a matching Designer.cs code file, so in this case we have UCLMain.Designer.cs. Now, if you were under the illusion that you were dragging real treeview and menu controls onto our form design and positioning them, and changing them, and that they really exist, then sorry – it's all a big scam!

All the IDE does when you drag a control onto a form is add some code to the Designer.cs file detailing how it should be drawn. Even though you create the form in a graphical way, C# actually draws the form on your screen from this file. It's another example of C# doing something in two differing ways in parallel, one that is better for itself, and one that is better for us.

So, in answer to some of the questions that will no doubt be swirling through your head:

- Can you edit this form directly? – Yes!
- Is it a good idea? – No!

Apart from the fact that designing a form in the IDE along with the property window is far easier to do, whenever you do change something in the IDE designer, C# will have a tendency to rewrite the Designer.cs file and wipe out your changes. This is fine if you are only doing things that are reflected in both the IDE property list and the Designer.cs form, it tends to leave these alone, but if you try and do something more advanced, like use a variable to store text – for multi language use, you may start to get very frustrated.

For now, let's just say that the only time we want to go into this file, is if we've made the mistake we did above and need to correct some disjointed code. Feel free to look through the Designer.cs file, you should notice that any controls we added to our form and any settings that we changed in the property window, will be here in this file.

Ok, so let's remove the offending line by simply deleting it, before we do though, can you see what it does? It assigns the event name `treeView1_AfterSelect` to the real event `AfterSelect` of the `treeView1` control so that we can then use this in our code. This allows C# to create events in your code for you under the controls name, so that they make sense to you, whilst linking the code you enter to the real events that control your form.

Now remove this line from the UCLMain.Designer.cs file and then save it.

Run UCL again, and you will have fixed your code.

Now let's do exactly the same thing again, and show you a way to fix this without having to manually edit the UCLMain.Designer.cs file. Before we do this though, you must have noticed that we are collecting more and more tabs in the main window display, some of which we don't really need to look at anymore.

When a tab is currently being displayed, you can click the "close" X at the right hand corner of the main window to close it. So while we're at it, let's close the UCLMain.Designer.cs file, and any others you feel you don't need open anymore. The more observant amongst you will also notice the down arrow next to the "close" X. This allows you to select any tabs that are open, but aren't visible in the main window. This will, of course, only occur if you have a lot of files open at the same time, but that does happen!

The only tabs I have open in the main window at the moment are the UCLMain.cs and UCLMain.cs[Design] files, but it is personal preference whether you like to keep your working area tidy or not.

Now, back to showing you a better way of rectifying any mistakes you may do with form events. Bring up your form design for UCLMain again and double click on the treeview control to bring up the same method once more.

First we still need to remove the newly created method in our UCLMain.cs file, we still need to do this because the IDE is a complete and utter coward and doesn't want to take the chance that it might be removing something you have typed in! After this, the way we should remove an associated event is to click on the events button, above the property list remember? Then find the "AfterSelect" event and clear out any text that is in there. This will remove the IDE's auto generated code from all of the right places, and also stop us fiddling around in Designer.cs files, which can actually stop your form design from being displayed in the IDE if you make a complete hash of it!

If you run UCL now, you will see that we didn't break it this time around!

Now, in the event list, find "DoubleClick" and double click on it for the IDE to create the method that we wanted in the first place. As you have probably figured out, unless you know which method the IDE will create automatically when you double click on a control in the design window, then it is less hassle to create new methods using the events list.

A point to note is that leaving an empty method and associated method event in your code will not do it any harm, as long as you don't delete one or the other, but your program will be processing stuff that it simply doesn't need to, which is just plain inefficient!

Now we should have this method prepared for us:

```
private void treeView1_DoubleClick(object sender, EventArgs e)
{

}
```

That's the right one! So we can get on with it, but first let's think about what we need to do next.

Can you see any potential problems here? We have gone through our XML file and populated our treeview in the UCLMain form, but we did nothing to read in the other values we store for the content, values like "ExeName", "ExePath" etc.

If we want to launch the content, then we need these values as well, so do we read them in every time we want to launch the content, or is there a better solution?

Well, reading all the values we need, every time we want to launch a program will be slow, and cause UCL to do a lot of file IO (input/output) operations, which are always worth avoiding if you can. As a basic rule of thumb, the more your application depends on opening files, then the more vulnerable it can become. A better choice would be to load all these values into memory, a thing that we can do, if you think about it, when we read the content list to populate the treeview control.

We can then simply read the memory location for the values we need whilst our program runs, and reading from memory is fast and won't rely on opening any more files to work. We already wrote the `LoadContentList` method in such a way that we can refresh our content list when anything changes, so this solution seems ideal, as it would also result in the corresponding memory values being refreshed at the same time, and therefore always being synchronised with the display.

So how do we get all that information into the computer's memory?

Well, didn't you notice that in the last tutorial you wrote that nice class called UCLContent, but we never even used it! Well, now is the time to use it! We're going to modify our code that reads the test xml file to also populate some memory locations with all the different values we will need to launch the code. To refresh your memory, here was our original code:

```
        private void LoadContentList()

          {

              if (File.Exists("UCLContent.xml"))
              {
                  XPathDocument doc = new XPathDocument("UCLContent.xml");
                  XPathNavigator nav = doc.CreateNavigator();

                  treeView1.Nodes.Clear();

                  TreeNode tNode;
                  string tType = "Application";
                  tNode = treeView1.Nodes.Add(tType);

                  XPathExpression expr;
                  expr = nav.Compile("/Content/" + tType + "/Name");
                  XPathNodeIterator iterator = nav.Select(expr);

                  try
                  {
                      while (iterator.MoveNext())
                      {
                          XPathNavigator nav2 = iterator.Current.Clone();
                          tNode.Nodes.Add(nav2.Value);

                      }
                  }
                  catch (Exception ex)
                  {
                      Console.WriteLine(ex.Message);
                      MessageBox.Show("Cannot load the UCL content list.
There is a problem processing the UCLContent.xml file!");
                  }
              }
              else
              {
                  MessageBox.Show("Cannot load the UCL content list. There is
no UCLContent.xml file present in the UCL folder!");
              }
          }
```

Now everything is fine with this method until we get here:

```
        tNode.Nodes.Add(nav2.Value);
```

Here, we add the name of our content into the treeview list, but don't store it in memory. We are going to need to create an object of type UCLContent in order to store all the information, and this is why our UCLContent class is designed to match the test xml file (or vice versa).

But there's another problem here, storing the information for a single content item is ok, but our final xml file will presumably contain lots and lots of content. How can we do this?

Well, there are a few ways to solve this problem in C#, but as this is a tutorial designed to teach you as much of C# as possible, and its fresh new ideas then let me introduce you to the ArrayList.

An ArrayList at its simplest level allows you to store a list of objects. These objects can be of any type, so it is perfectly acceptable to have an ArrayList of type UCLContent. That seems ideal to me, and just what we are looking for!

To make an ArrayList of type UCLContent we can use the following:

```
ArrayList UCLContentList = new ArrayList();
```

This would allow for multiple objects of any type to be stored in an ArrayList called `UCLContentList`. However, as we will need `UCLContentList` to be available throughout the code in our form, meaning we will want to add items to it in the `LoadContentList()` method, but would also need to be able to read the list from different methods, we need to declare the ArrayList in a special place for it to be useable like this.

When you declare an object inside a method and populate it, then only that method can see the object and use it. If you were to then declare the same object in another method, even if you called it the same name, it would not contain the information we set in the first method.

We need this object; that will hold all the information about our list, to be shared between all the methods in our form. We do this by adding it to the code outside of any methods, but still within the class, so directly beneath the lines:

```
public partial class UCLMain : Form
{
```

Insert the line:

```
private ArrayList UCLContentList = new ArrayList();
```

This will create the ArrayList `UCLContentList` that can be used from within any of our methods in the UCLMain form. Because the ArrayList object is located within the framework collections namespace, then we also need to add the following `using` statement at the top of our code:

```
using System.Collections;
```

Now we have an object in place to pass values to from the test XML file as we read it, let's put in the code to actually do this. At the moment, within the `while` loop of the `try` statement, we have the following lines of code:

```
XPathNavigator nav2 = iterator.Current.Clone();
tNode.Nodes.Add(nav2.Value);
```

As explained earlier, these lines just put the description of the content into our treeview control, but we also need to put this description into our `UCLContentList` ArrayList.

Now, consider if we create a temporary object of type `UCLContent` called `tContent` that we can populate with the values from our test XML file before adding it to the `UCLContentList` ArrayList. The reason we would do this, is that an ArrayList can contain any type of object; it is not just limited to a single type. So we can define a temporary object of type `UCLContent`, and then simply add that to our ArrayList to build it item by item. Later, if we want to add a different object, we can define another temporary object for whatever type that is, and add it to the same ArrayList.

Since we are creating a temporary object called `tContent` of type `UCLContent`, it would be nice if we could assign values to it like this:

```
tContent.Name = nav2.Value;
```

Well, we can do this in C#, but if you try it now, then you will get an error, and this is due to the way we designed our UCLContent class way back in tutorial two – I told you that this class was going to be trouble!

Part Two: Redefining the UCLContent class

Let's look at the design of our class from way back in tutorial two:

```
UCLType Type;
string Name;
string Icon;
string ExeName;
string ExePath;
string FileName;
string FilePath;
string Arguments;
```

What we actually did here was declare some `string` variables and a type (`enum`) object that we need to store details of our content information, by declaring them in this manner, we only created localised variables within the UCLContent class, and so none of these can actually be read or set from outside of our UCLContent class.

What a dilemma! What's the point in writing a class if we can't use it!

What we need are local variables for the class like these, for use internally by methods in our class itself, but also a way to expose them to other code that wants to use them, that isn't located within the UCLContent class, so code like our `LoadContentList()` method that is running from the UCLMain class. Now before you run off and type `public` in front of all of them, which you may have figured out its meaning, there is an even better way to do this.

Luckily C# is able to cope with this and allows us to setup an easy way of retrieving and setting values for our class variables by using properties. Think of a property as a way of getting or setting the value of a class variable from outside of the class.

As the names we assigned to these variables is really what we would like to use externally, let's rename them all to something different, like so:

```
UCLType mType;
string mName;
string mIcon;
string mExeName;
string mExePath;
string mFileName;
string mFilePath;
string mArguments;
```

Now, to expose the variable to the outside world we create a property like so:

```
public UCLType Type
 {
     get
     {
         return this.mType;
     }
     set
     {
         this.mType = value;
     }
 }
```

As you can see, this property is called by the name `Type`, and has a `get` and a `set` method to allow us to `get` or `set` the underlying local variable in the class. Notice we declare the property as `public` so that external code can use it. Now, an important thing to note about the `Type` property above is that we originally created an `enum` for it. As this is the case, then we must also make sure that the `enum` declaration is `public` to avoid an error when we try and access the `Type` property from outside of the UCLContent class.

The really neat thing about this is that whenever we type in the name of an object that is of type `UCLContent` in the IDE, then after typing in a "." our properties will appear in the IDE's dropdown context list, allowing easy selection of our custom properties!

The really upsetting thing about this is that we need to do this for each variable that we want a property for, so get to it!, copy and paste is your friend, and at the end you should end up with a UCLContent class that looks like this:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace UCL
{
    class UCLContent
    {

        public enum UCLType
        {
            Game = 0,
            Application = 1,
            Video = 2,
            Audio = 3,
        }

        UCLType mType;
        string mName;
        string mIcon;
        string mExeName;
        string mExePath;
        string mFileName;
        string mFilePath;
        string mArguments;

        public UCLType Type
        {
            get
            {
                return this.mType;
            }
            set
            {
                this.mType = value;
            }
        }

        public string Name
        {
            get
            {
```

```
            return this.mName;
        }
        set
        {
            this.mName = value;
        }
    }

    public string Icon
    {
        get
        {
            return this.mIcon;
        }
        set
        {
            this.mIcon = value;
        }
    }

    public string ExeName
    {
        get
        {
            return this.mExeName;
        }
        set
        {
            this.mExeName = value;
        }
    }

    public string ExePath
    {
        get
        {
            return this.mExePath;
        }
        set
        {
            this.mExePath = value;
        }
    }

    public string FileName
    {
        get
        {
            return this.mFileName;
        }
        set
        {
            this.mFileName = value;
        }
    }

    public string FilePath
    {
        get
        {
            return this.mFilePath;
```

```
            }
            set
            {
                this.mFilePath = value;
            }
        }

        public string Arguments
        {
            get
            {
                return this.mArguments;
            }
            set
            {
                this.mArguments = value;
            }
        }

    }
}
```

Our initial stab at the UCLContent class was just too good to be true in tutorial two! Hey that even rhymes!

Now we have modified our UCLContent class to allow the reading and writing of its property values, we can start to finish off the `LoadContentList()` code.

Sadly, we are now about to embark on another hard hitting feature of C#, so I'm just going to paste in the new code and go through it again. If you're new to programming, then this will melt your mind, sorry, but maybe I can at least make it more understandable:

```
        UCLContent tContent = new UCLContent();

        XPathNavigator nav2 = iterator.Current.Clone();
        Type type = tContent.GetType();
        PropertyInfo tTemp = type.GetProperty(nav2.Name);
        tTemp.SetValue(tContent, nav2.Value,null);
        tNode.Nodes.Add(tContent.Name);

        tTemp = type.GetProperty("Type");
        UCLContent.UCLType tUCLType =
(UCLContent.UCLType)Enum.Parse(typeof(UCLContent.UCLType), tType, true);
        tTemp.SetValue(tContent, tUCLType , null);

        while (nav2.MoveToNext())
        {
            tTemp = type.GetProperty(nav2.Name);
            tTemp.SetValue(tContent, nav2.Value,null);
        }

        UCLContentList.Add(tContent);

        tContent = null;
```

Remember that this code is contained within the original `while` loop in our `LoadContentList()` method.

The first line is where we declare our temporary object `tContent` as type `UCLContent.`

```
UCLContent tContent = new UCLContent();
```

The second line is the same as it always was, but then it all goes downhill rapidly! As we already know what the second line does, let's take it from the third line, but first I want to try and explain to you why we have had to write all of the code listed above.

A few pages back, I said it would be nice if we could write a line like:

```
tContent.Name = nav2.Value;
```

Well, now that we have created all those properties in our UCLContent class, we can indeed do this, but there is another problem. Whilst it is safe to assume that all our content has a "Name", it may not all have an "Argument" value, and indeed our test XML file only has "Name", "ExeName" and "ExePath" included. We therefore need some way of setting a property dependent upon its name, so if we read in an XML element called "ExeName", we want to make sure we set the "ExeName" property in our `tContent` object.

That's pretty easy to understand, but not easy to do in C#. Visual Basic and Java coders the world over will be laughing out loud now and shouting "eval" in your face, as this is easily handled in those languages, but I'm afraid we've lucked out on this one in C#. We can still do it, but to do it in C#, we need to use a very powerful and complex feature called Reflection. Let's not try and describe completely what Reflection does, it will probably confuse me, and it goes far, far deeper than an "eval" function ever would! But essentially what we are trying to do is something like:

```
tContent.(We need to use a text string to pass the property name) =
nav2.Value;
```

The problem in C# is that unlike VB and Java, where you could simply do something similar to this:

```
tContent.eval("ExeName") = nav2.Value
```

Which takes the `eval` statement and evaluates it to get its true definition. So in this example VB and Java would see that line of code as this:

```
tContent.ExeName = nav2.Value
```

We simply don't have this in C#. But reflection lets us achieve a similar thing, just in a far more complicated manner! Now, let's go through the code and see how it works, but first, to use Reflection, we will need yet another `using` statement at the beginning of our code:

```
using System.Reflection;
```

Now let's go through the rest of the method, line by line:

```
Type type = tContent.GetType();
```

Firstly, we are simply finding the type of the object that we will be updating, in this case the object is `tContent` so the type will be `UCLContent`.

```
PropertyInfo tTemp = type.GetProperty(nav2.Name);
```

Ah, now `PropertyInfo` is the heart behind this reflection action, and what we have done here is set a temporary object called `tTemp` to the property of the type `UCLContent` that is named `nav2.Name`, so in this case, we have simply grabbed the property "Name" as `nav2.Name` currently contains "Name".

```
tTemp.SetValue(tContent, nav2.Value, null);
```

And now in this line, we have set the value of that property to `nav2.Value`, which in this example is "Microsoft Notepad", the same value that we use in our treeview display. For this to work, in the `SetValue` method, we have to declare the underlying object variable, `tContent`, the value we want to give to it (`nav2.Value`) and also its index, which is not required here so we can simply pass `null`.

So now our `tContent` variable has the "Name" property set, and to prove it we can use that to populate our treeview display:

```
tNode.Nodes.Add(tContent.Name);
```

Now we need to do something slightly different to set the Type value of `tContent`, as this is not a simple `string` variable, but an `enum` that we defined ourselves:

```
tTemp = type.GetProperty("Type");
UCLContent.UCLType tUCLType =
(UCLContent.UCLType)Enum.Parse(typeof(UCLContent.UCLType), tType, true);
tTemp.SetValue(tContent, tUCLType , null);
```

The first line should be known to you now, and the second line in this code is addressing the issue of getting the value for the `enum`, by looking up its name. We do this by using a special `Enum` type that can be parsed from the text of the `enum` name into the value of the `enum`, this is very complex, and we won't dwell on it here, but after this is done we can then simply assign it to our property like we did for the "Name" value.

And now it starts to get a bit more interesting, if you haven't nodded off already! Hopefully you will recall that our original `iterator` was set to search the test xml file by the type "Application" and retrieve the element "Name" to populate the treeview list. We were able to read this "Name" by creating a clone record from the `iterator` called `nav2`.

As `nav2` is a navigator itself then we can use this to step through all the child elements of the "Application" element, so we can retrieve not just the "Name", but also the "ExeName", "ExePath" and so on, just by moving through the navigator `nav2`.

```
 while (nav2.MoveToNext())
{
```

Here we go, for each additional element (other than “Name”) in the `nav2` navigator:

```
   tTemp = type.GetProperty(nav2.Name);
   tTemp.SetValue(tContent, nav2.Value,null);
}
```

Get the property again, but this time we will get what is in `nav2.Name`, so we will be looking at properties other than “Name”. And when we have grabbed that property, we can then write to it in exactly the same way that we wrote “Name” and “Type”. The `while` loop will then continue until we have read all the elements for this “Application” record. The really neat thing about this is that it doesn’t matter how many of the elements we populate for each content item in the list, this code will handle it if we only have “ExeName” and “ExePath” populated, or if we have “ExeName”, “ExePath” and “Argument” populated!

And then:

```
UCLContentList.Add(tContent);
```

When we have filled the `tContent` object with all the information from the XML file, we can write it into our `UCLContentList` ArrayList, we can keep on adding to the ArrayList as many times as we want, thus building our list of content.

Lastly:

```
tContent = null;
```

An important thing to remember about C# is that each time you create a new object, no matter what its type; it will use up some of your computer’s memory. We need to ensure that we dispose of the object after use, and can do that by setting it to `null` when we have finished with it.

Now, you may be wondering why I haven’t mentioned this before, and some of you may even have heard about C#’s garbage collection features, which allow it to free up memory by disposing of the variables that you forgot to, automatically. While this is great, we sometimes might want to force C# to dispose of our variables from within our code, particularly if we are creating them in a loop, which may lead to problems as the loop progresses, before C# has a chance to automatically address them. To be honest, this loop is too small to pose a problem, but it’s something worth noting for the future.

Therefore we create a new instance of our `UCLContent` object at the start of the loop, and dispose of it at the end. This is not always necessary, but as memory leaks in loops are awkward to find and vary from machine spec to machine spec, it’s a good habit to get into.

Some may say why not do this for all objects then, and indeed you could, if you wanted to make sure you have full control of memory allocation, rather than relying on C# to do the work for you.

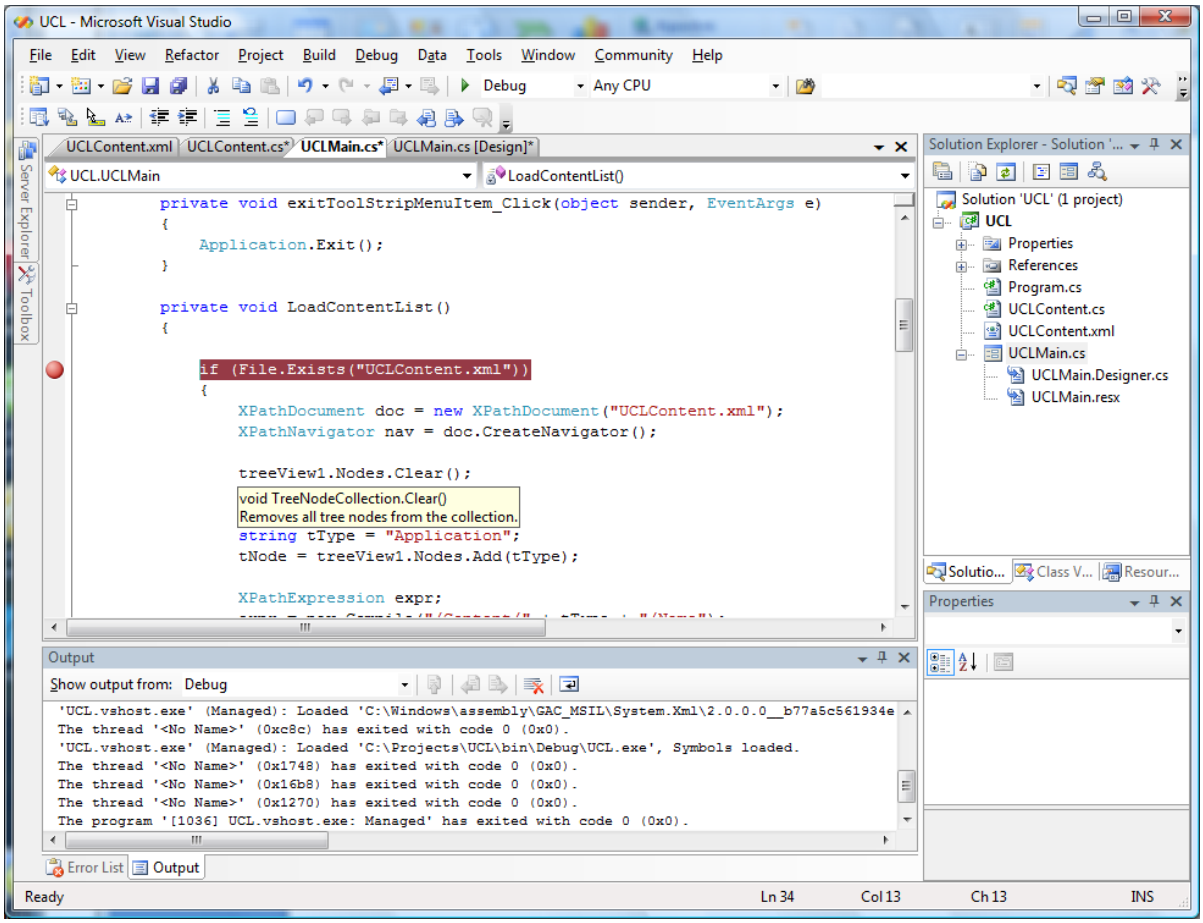
Here's the final code for the `LoadContentList()` method:

```
        private void LoadContentList()
        {
            if (File.Exists("UCLContent.xml"))
            {
                XPathDocument doc = new XPathDocument("UCLContent.xml");
                XPathNavigator nav = doc.CreateNavigator();
                treeView1.Nodes.Clear();

                TreeNode tNode;
                string tType = "Application";
                tNode = treeView1.Nodes.Add(tType);

                XPathExpression expr;
                expr = nav.Compile("/Content/" + tType + "/Name");
                XPathNodeIterator iterator = nav.Select(expr);

                try
                {
                    while (iterator.MoveNext())
                    {
                        UCLContent tContent = new UCLContent();

                        XPathNavigator nav2 = iterator.Current.Clone();
                        Type type = tContent.GetType();
                        PropertyInfo tTemp = type.GetProperty(nav2.Name);
                        tTemp.SetValue(tContent, nav2.Value,null);
                        tNode.Nodes.Add(tContent.Name);

                        tTemp = type.GetProperty("Type");
                        UCLContent.UCLType tUCLType =
(UCLContent.UCLType)Enum.Parse(typeof(UCLContent.UCLType), tType, true);
                        tTemp.SetValue(tContent, tUCLType , null);

                        while (nav2.MoveToNext())
                        {
                            tTemp = type.GetProperty(nav2.Name);
                            tTemp.SetValue(tContent, nav2.Value,null);
                        }

                        UCLContentList.Add(tContent);

                        tContent = null;
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                    MessageBox.Show("Cannot load the UCL content list.
There is a problem processing the UCLContent.xml file!");
                }
            }
            else
            {
                MessageBox.Show("Cannot load the UCL content list. There is
no UCLContent.xml file present in the UCL folder!");
            }

        }
```

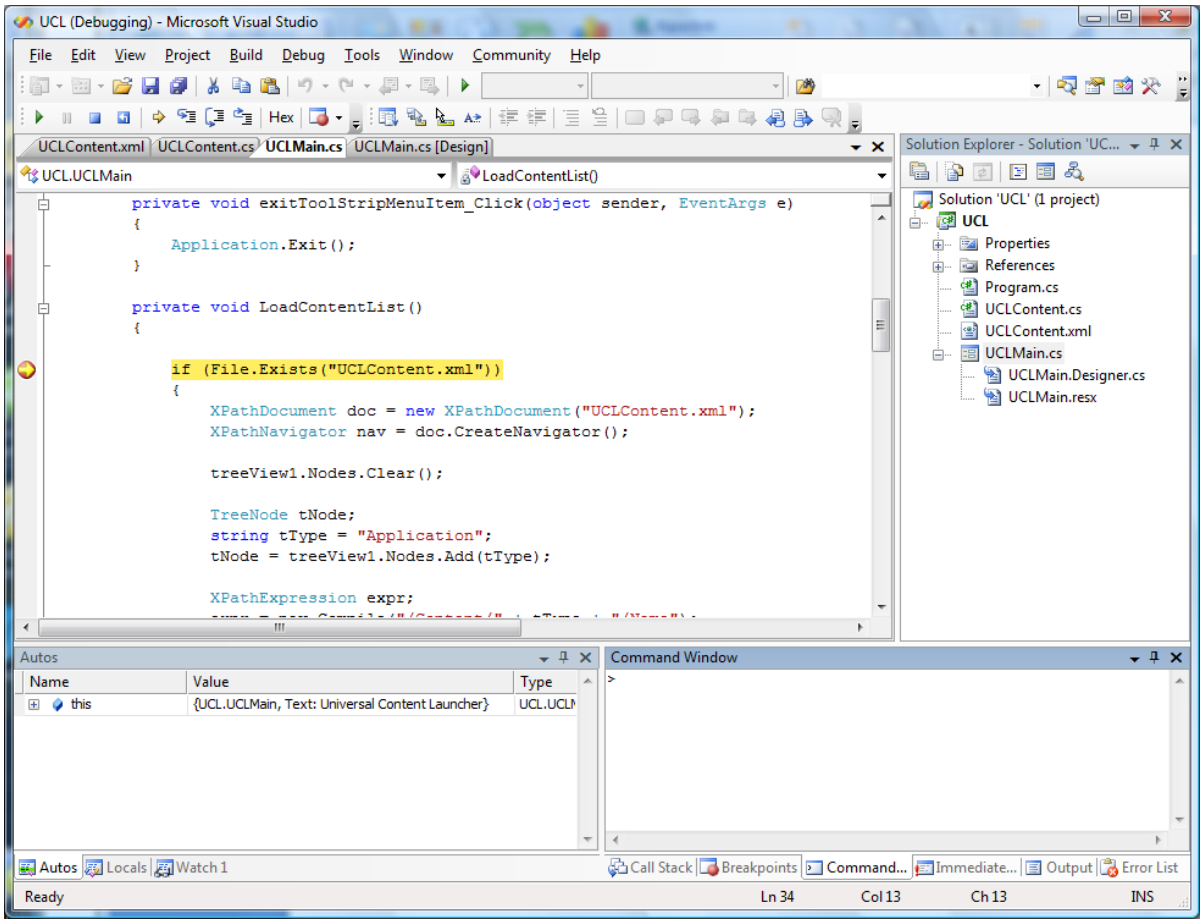
Part Three: Debugging your code

All that is pretty frightening, but don't be scared, we are in control here and I'll prove it!

Enter all the code above so your `LoadContentList()` method is the same. Now in the main window of your IDE, whilst viewing the `LoadContentList()` method, click in the column to the left of the code, next to the "File.Exists" line, you will see a red blob like so: (the red blob in this screenshot is exactly where you should be clicking)



What you have just done is set a breakpoint in the code. A breakpoint is something that will freeze your code at that exact point when your application tries to execute it. Now go ahead and run UCL, your code should stop at the breakpoint, like so:



Make no mistake, your code is still running, but we have simply stopped it at this point, this will do no harm, and your application is just sat there waiting to continue, but it gives us the opportunity to step through our code and even look at the values that different variables and objects are holding at this exact point in time.

To step through the code press F8 on your keyboard (or F10 if F8 doesn't work)

You will notice that the yellow line moves to the next line of code. Oh my god, we made UCL process a single line of code, it is our slave now! Keep stepping through the code until you get to the line:

```
UCLContent tContent = new UCLContent();
```

So that it is highlighted in yellow, now hold your mouse over `tContent`, and you will see the following in a little popup box:

```
tContent | null
```

Now press F8\F10 to step forward again, so the yellow highlight is on the line of code under this one, and move your mouse over `tContent` in the above line once more, now it is this:

```
+ | tContent | {UCL.UCLContent}
```

Now move over the "+", and you will see a lot of fields that directly relate to your UCLContent class. What has happened is that the variable `tContent` has now initialized itself as a new

object for the type `UCLContent`, and these are the default values for our various properties.

Keep going – this is getting good! Step through all the lines and move your mouse over variables and objects to see how their values change, hover over the "+" symbols to drill deeper into the objects. You will see the `UCLContentList` ArrayList gradually fill with the data from our test XML file!

The thing to remember here is that the line of code that is highlighted has not been processed yet, you need to step over the line you want to interrogate. Once you get bored of this you can press F5, and this tells C# to now run until the next breakpoint, and as there isn't one, our program will run as normal.

Once you have finished messing around then click on the red breakpoint blob to remove it again, if we don't do this, then our code will always stop at the breakpoint when we run UCL.

How handy is that? If you have a bug in your program and don't quite know what is happening in a method or piece of class code, then just set a breakpoint, put on your Sherlock Holmes cap, and away you go!

Part Four: Ready to Launch

Could it be that we have actually reached the point where we can write our `treeView1_DoubleClick` method? It's been a long day!

Well we are here now, so let's get to it: first we need to write a method that can actually do the job of launching the content. The easiest place to do this is in the UCLContent class. This means that if we write a Launch method here, then if we have a variable of type `UCLContent` called `tContent` then we can simply do this:

```
tContent.Launch();
```

Now that seems quite easy, and it has been a long day, so let's not debate this further, here's the method that you need to add to your UCLContent class file:

```
        public Boolean Launch()
        {

            try
            {

                System.IO.Directory.SetCurrentDirectory(this.mExePath);
                System.Diagnostics.Process.Start(this.mExePath +
this.mExeName, this.mArguments);
                return true;

            }

            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                return false;
            }

        }
```

This method, called Launch(), returns a value of type `Boolean`, so it will either be true or false depending on whether it failed or succeeded. Notice once again we use the `try` statement to catch any errors we may have with the `catch` statement following it.

The return value of the method is set according to whether the `try` statement succeeds or fails with the `return` setting.

Because this method returns a `Boolean` then the following C# statement will be valid:

```
if (tContent.Launch() == true)
```

Notice how this line:

```
System.IO.Directory.SetCurrentDirectory(this.mExePath);
```

Has the full reference to the `Directory` method as we haven't included the `using System.IO;` statement at the top of our code, if we had, then we could have written it like this:

```
Directory.SetCurrentDirectory(this.mExePath);
```

This has been done this way purely to offer an example of how to code without using the `using` statement. If you want to change it then please feel free to do so, I probably will before the end of this tutorial!

Lastly the line:

```
System.Diagnostics.Process.Start(this.mExePath + this.mExeName,
this.mArguments);
```

This is the line that actually launches our content. This is a very simple use of the `System.Diagnostics.Process` object, and we will probably expand this more at a later date. Notice how we use the UCLContent class local variables, mExeName and mExePath in this code as opposed to the properties, as we are running from within the class itself.

Note also that because the `System.Diagnostics.Process` method allows us to pass arguments to the content we will run, we have made use of this also, as it will not be affected by passing the `mArguments` variable if it is empty for the chosen Content.

After you have put this method into your UCLContent class, we simply need to call if from the `treeView1_DoubleClick` method. But how can we do this?

We have created the Method Launch() for our UCLContent class, so that we can do the following:

```
tContent.Launch()
```

But in order to do this we need to set `tContent` to the `UCLContent` item in the `UCLContentList` ArrayList that matches the `treeview` node name that we double clicked on! Get It?

Well, you may have already guessed it, but we need to write a quick method to return an object of type `UCLContent` that exists in `UCLContentList` simply by comparing the name that we supply it. Here it is, see if you understand it before I explain it:

```
private UCLContent GetContentFromName(string NameToCheck)
{
    foreach (UCLContent tContent in UCLContentList)
    {
        if (tContent.Name == NameToCheck)
        {
           return tContent;
        }
    }

    return null;

}
```

Firstly, notice that this method is not declared as void, so it returns something, and in this case we are saying that it returns an `UCLContent` object.

Second, notice that we called our method `GetContentFromName` and then in brackets after that declared a new variable `NameToCheck` of type `string`.

Both of these things are significant. First, if this method returns a valid type object for `UCLContent`, then we must call it in a way that the returned object can be used correctly. See the line below in a moment.

Second, because the method specifies a value that it needs to use, then we must supply one to prevent the call from generating an error.

We would therefore do something like this in our `treeView1_DoubleClick` method:

```
            UCLContent tContent =
GetContentFromName(treeView1.SelectedNode.Text);
            tContent.Launch();
```

Looking at this you can see we create a new `UCLContent` object called `tContent`. This will then become the `UCLContent` object that the `GetContentFromName` method returns. We have passed the `string` that is contained in `treeView1.SelectedNode.Text` to the method, which is the name of the content. As we used the same name in populating the treeview and filling our ArrayList, then we know that this will match.

The next line simply launches the content, as we can now do this when we have a valid `UCLContent` object to launch from.

Looking at the `GetContentFromName` method once more, if you analyse it you will see that it does little more than search through all the `UCLContent` objects in our `UCLContentList` ArrayList, and return the one whose name matches.

There are a few things to note in this simple method:

You can use the `foreach` statement to loop through all the objects in an ArrayList

Look at the `if` statement, it contains not one but two "=" characters. This is because C# will recognise one "=" as something should be set to the specified value, and two "==" characters as conditionally querying if something is equal to the specified value. The conditional part of an `if` statement must always be enclosed in brackets, and the code that follows when the condition is met must be enclosed in braces. So, if in this case the Content Name we are currently looking at in the loop, matches the Content Name we are querying, then the method will return that content object.

If no Content matches the Name we supplied, then this method will return `null`, which will allow us to trap any invalid content before we attempt to launch it, in our final method for `treeView1_DoubleClick` like so:

```
        private void treeView1_DoubleClick(object sender, EventArgs e)
        {

            UCLContent tContent =
GetContentFromName(treeView1.SelectedNode.Text);
            if (tContent != null)
            {
                tContent.Launch();
            }
        }
```

Note that this `if` statement uses the condition "!=" which means Not Equal To. So as long as the method `GetContentFromName` returns a valid `UCLContent` object, then it will be launched.

You could easily modify the methods above to check for a valid file path and executable name for the content before launching it, and display an appropriate message to the user if it cannot be found, couldn't you? You could also use more `using` statements to make the code above clearer, and even display a message box if the `GetContentFromName` method fails to find the content.

I think that I will leave that up to you, during these tutorials, you may see that I have modified these methods to be more friendly for the end user, we might even end up both doing the same changes. Why don't you try and add in some of these things, and keep an eye out for my changes in the next tutorial.

That's it for tutorial three, you really are doing great! Look at all that you have done so far, and to think it wasn't that long ago that you found the prospect of programming anything in C# daunting, let alone an application like this.

Congratulations, take a well earned rest, in tutorial four, we will be building further on what we have discussed here.